

Research - Kasteran* — Programming Language Design Principles

Alpasan, Lois-Kleinner

2026

Abstract

Programming language design is a discipline spanning formal semantics, human-computer interaction, compiler engineering, and software engineering methodology. This document surveys the foundational principles of language design—syntactic abstraction, domain-specific languages (DSLs), cognitive dimensions, and the trade-offs between expressiveness and safety—drawing on the work of Felleisen, Krishnamurthi, Steele, and others. We examine how these principles inform Kasteran*'s design, from its rune-based syntax to its gradual type system and linear capability model.

Kasteran* — Programming Language Design Principles

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

Abstract

Programming language design is a discipline spanning formal semantics, human-computer interaction, compiler engineering, and software engineering methodology. This document surveys the foundational principles of language design—syntactic abstraction, domain-specific languages (DSLs), cognitive dimensions, and the trade-offs between expressiveness and safety—drawing on the work of Felleisen, Krishnamurthi, Steele, and others. We examine how these principles inform Kasteran*'s design, from its rune-based syntax to its gradual type system and linear capability model.

1. Introduction

The design of a programming language is, in Guy Steele's words, "the most important tool a programmer has" (Steele 1). Language design decisions reverberate through the entire software ecosystem, affecting programmer productivity, code quality, safety, performance, and maintainability for decades. Kasteran* is designed with explicit attention to language design principles, balancing competing goals—safety vs. expressiveness, simplicity vs. power, compile-time checking vs. runtime flexibility—in a coherent, principled manner.

2. Historical Background

The modern understanding of programming language design emerged from the synthesis of several intellectual traditions. Algol 60 established the tradition of formal syntax description using BNF (Backus-Naur Form), enabling precise communication about language structure (Naur et al. 1). Simula introduced object-oriented programming concepts (classes, objects, inheritance) that would

dominate mainstream language design for decades (Dahl and Nygaard 1). Lisp introduced the radical idea that programs are data structures (S-expressions), enabling homoiconic metaprogramming and macro-based syntactic abstraction (McCarthy 1).

The Language Design Principles community, centered around the seminal work of Felleisen, Krishnamurthi, and Findler, established a design methodology based on the “programming language as a tool for thought” perspective (Felleisen and Krishnamurthi 1). Their work on DrScheme (now Racket) demonstrated that a language could be designed for progressive understanding: beginners start with a subset, gradually learning more advanced features as their programming skills grow. The “How to Design Programs” (HTDP) methodology emphasizes systematic program design based on data-driven decomposition (Felleisen et al. 1).

Matthias Felleisen’s work on syntactic abstraction and macros established the theoretical foundations of hygienic macro systems, where macros preserve lexical scoping and prevent unintended variable capture (Felleisen 1). The Racket language, built on PLT Scheme, demonstrated the power of a language-oriented programming model where the programmer can create and compose DSLs as easily as functions.

Guy Steele’s “Growing a Language” talk at OOPSLA 1998 articulated a vision of languages that can be extended by their users (Steele 1). Steele argued that a language should support organic growth: new constructs can be added by library authors, not just by language designers. This philosophy influenced the design of Java (with its library-centric model) and the metaprogramming capabilities of modern languages.

Shriram Krishnamurthi’s work on the “automaton of programming language research” articulated the design space of programming languages in terms of “features” (independent, composable units of functionality) (Krishnamurthi 1). His work demonstrates that language design can be studied empirically through controlled experiments on programmer behavior, challenging the folklore-based approach of earlier decades.

The Cognitive Dimensions of Notations framework, developed by Green and Blackwell, provides a structured approach to evaluating language usability (Green and Petre 1). Key dimensions include: viscosity (resistance to change), hidden dependencies (connections between distant code locations), secondary notation (formatting and comments), premature commitment (order of decisions), and closeness of mapping (correspondence between problem and notation).

3. Technical Analysis

Kasteran*’s design is guided by several explicit principles derived from this literature.

Principle 1: Syntactic Abstraction through Hygienic Macros. Kasteran* provides a pattern-based macro system with lexical scoping guarantees. Macros are first-class: a macro can be imported from a library, exported from a module, and composed with other macros. The macro expander operates on the typed AST, not the raw token stream, preventing the token-level errors that plague C preprocessor macros.

Principle 2: Gradual Typing with Safe Defaults. The type system defaults to static checking for all function boundaries and module interfaces. Within a function body, the programmer can annotate types for stronger checking or omit types for inference-guided development. The “safe defaults” principle means that the default compilation mode enforces memory safety (through linear types) and data-race freedom (through capabilities). The programmer must explicitly opt

into unsafe features through `unsafe` annotations.

Principle 3: Language-Oriented Programming. Kasteran* supports the creation of embedded DSLs through its macro system, literate syntax extensions, and type system. A DSL in Kasteran* is a subset of the language with its own syntax (defined through macros) and its own type rules (defined through type system extensions). The type system ensures that DSL code is type-safe and that errors are reported in terms of the DSL’s concepts, not the underlying implementation.

Principle 4: Progressive Disclosure. Kasteran* is designed to be learnable in stages. The basic syntax (variables, functions, control flow) requires minimal rune knowledge. The macro system, linear types, capabilities, and verification features are incremental layers that the programmer adopts as needed. The IDE (Kasteran* Studio) provides feature levels that show or hide advanced constructs based on the programmer’s experience level.

The formal semantics of Kasteran* are defined operationally through a reduction semantics (Felleisen-style PLT Redex model). The semantics cover the core calculus (Kaster, a linear lambda calculus with capabilities), the macro expansion phase (a syntactic theory of hygienic macro expansion), and the effect system (a small-step operational semantics for capability tracking). The formal semantics serve as the specification for the compiler implementation and as the basis for the metatheoretic proofs of type safety and memory safety.

4. Current State of the Art

Modern language design research focuses on several frontiers. The Rust language has demonstrated that ownership-based memory safety is practical for systems programming, influencing the design of future languages (including Kasteran*, Vale, Hare, and Zig). The Go language’s minimalism celebrated the virtues of simplicity in language design, although its lack of generics (addressed in Go 2.0) and limited expression caused frustration for many programmers.

The Julia language demonstrated that a dynamic, multiple-dispatch language could achieve performance competitive with statically-typed systems languages through just-in-time compilation and type specialization. Julia’s design—combining the ergonomics of Python with the performance of C—influences Kasteran*’s approach to gradual typing and compilation.

The Racket language continues to advance language-oriented programming, with the “Racket Manifesto” articulating a vision of programming as language design (Felleisen et al. 1). Racket’s `#lang` mechanism enables any module to define a new language, with its own syntax, semantics, and tools. This approach inspires Kasteran*’s modular language extension system.

5. Relevance to Kasteran*

Kasteran* synthesizes the principles surveyed above into a coherent design. The language is designed for three primary use cases: systems programming (operating systems, device drivers, embedded systems), high-performance computing (scientific simulation, data processing, machine learning), and full-stack development (web applications through Wasm compilation).

The design process followed the HTDP methodology: we began with a core language (Kaster), defined its semantics formally, implemented the core compiler, and then added features incrementally, verifying at each step that the new features did not compromise the core safety guarantees.

The macro system follows the “specifications for a hygienic macro system” established by the Scheme community (Kohlbecker et al. 1). The type system follows the “safe, fast, and easy” design

principle of Rust, with extensions for fractional capabilities and gradual typing. The effect system draws on Koka’s algebraic effect handlers and the Prior work on effect polymorphism.

6. Future Directions

The future of programming language design lies in greater integration with formal verification. Languages like Dafny, F, and *Lean* demonstrate that verification can be integrated into a general-purpose programming language without sacrificing usability. Kasteran’s SMT-backed verification pipeline is a step in this direction.

Another direction is “conversational language design”: using machine learning and program synthesis to co-design languages with their communities. Large language models can analyze language usage patterns and suggest design improvements, while program synthesis can automatically generate implementations from design specifications. The intersection of PL design and AI is an emerging research area with significant potential.

Works Cited

- Dahl, Ole-Johan, and Kristen Nygaard. “SIMULA: An ALGOL-Based Simulation Language.” *Communications of the ACM*, vol. 9, no. 9, 1966, pp. 671-678.
- Felleisen, Matthias. “On the Expressive Power of Programming Languages.” *Science of Computer Programming*, vol. 17, no. 1-3, 1991, pp. 35-75.
- Felleisen, Matthias, and Shriram Krishnamurthi. “The DrScheme Project.” *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 1-10.
- Felleisen, Matthias, et al. *How to Design Programs: An Introduction to Programming and Computing*. 2nd ed., MIT Press, 2018.
- Felleisen, Matthias, et al. “The Racket Manifesto.” *Proceedings of the 2015 Summit on Advances in Programming Languages*, 2015, pp. 1-15.
- Green, Thomas R. G., and Marian Petre. “Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Approach.” *Journal of Visual Languages and Computing*, vol. 7, no. 2, 1996, pp. 131-174.
- Kohlbecker, Eugene, et al. “Hygienic Macro Expansion.” *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986, pp. 151-161.
- Krishnamurthi, Shriram. “The Automaton of Programming Language Research.” *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 1-12.
- McCarthy, John. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.” *Communications of the ACM*, vol. 3, no. 4, 1960, pp. 184-195.
- Naur, Peter, et al. “Revised Report on the Algorithmic Language ALGOL 60.” *Communications of the ACM*, vol. 6, no. 1, 1963, pp. 1-17.
- Steele Jr., Guy L. “Growing a Language.” *Higher-Order and Symbolic Computation*, vol. 12, no. 3, 1999, pp. 221-236.

- Abelson, Harold, and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed., MIT Press, 1996.
- Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.
- Harper, Robert. *Practical Foundations for Programming Languages*. 2nd ed., Cambridge University Press, 2016.
- Flatt, Matthew, and PLT Group. “Reference: Racket.” *PLT Inc., Technical Report PLT-TR-2010-1*, 2010.
- Tarditi, David, et al. “No Assembly Required: Compiling Standard ML to C.” *ACM Letters on Programming Languages and Systems*, vol. 1, no. 2, 1992, pp. 161-177.
- Hudak, Paul. “Conception, Evolution, and Application of Functional Programming Languages.” *ACM Computing Surveys*, vol. 21, no. 3, 1989, pp. 359-411.
- Cardelli, Luca. “Typeful Programming.” *IFIP State-of-the-Art Reports*, Springer, 1991, pp. 1-52.
- Meyer, Bertrand. *Object-Oriented Software Construction*. 2nd ed., Prentice Hall, 1997.
- Cunningham, Ward, and Kent Beck. “A Diagram for Object-Oriented Programs.” *Proceedings of the 1986 OOPSLA Conference*, 1986, pp. 1-8.
- Martin, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- Sussman, Gerald Jay, and Guy L. Steele Jr. “Scheme: An Interpreter for Extended Lambda Calculus.” *AI Memo 349, MIT Artificial Intelligence Laboratory*, 1975.
- Reynolds, John C. “Definitional Interpreters for Higher-Order Programming Languages.” *Proceedings of the ACM Annual Conference*, 1972, pp. 717-740.
- Wadler, Philip. “The Essence of Functional Programming.” *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1992, pp. 1-14.